Abstract

- A scalable **distributed file system** for large **distributed data-intensive applications**.
- Fault rolerance

Introduction

performance, scalability, reliability, and availability.

Different points int the design space

- component failures are the norm rather than the exception.
    - The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures.Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.
- As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.
- Third, **most files** are mutated by **appending new data** rather than overwriting existing data. Once written, the files are only read, and often only sequentially. also introduced an atomic append operation.
- Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility

Design Overview

Assumption

- The system is built from components that **often fail**. It must constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis.
- We expect a few million files, **typically 100 MB or larger** . Multi-GB files are the common case and should be managed **efficiently** Small files must be supported, but we need not optimize for them.
- The workloads primarily consist of two kinds of reads:**large streaming reads(1MB or more)** and **small random reads(few KBs at some random offset, often sort the reads by the offset to prevent go back and forth)**.
- The workloads also have many large, sequential writes that append data to files. size are close to reads. Once written, files are **seldom modified again**. Small writes at arbitrary positions in a file are supported but **do not have to be efficient**.
- system must efficiently implement **well-defined semantics** for **multiple clients that concurrently** append to the same file.
- High sustained bandwidth is more important than low latency.
    - 带宽（Bandwidth）是指**在单位时间内可以传输的数据量**。
      延迟（Latency）则是指数据从**发送端到接收端所经历的时间延迟**。

Interface

Files are organized hierarchically in directories and identified by pathnames.

`create` `delete` `open` `close` `read` `write` `snapshot`

- **Snapshot** creates a copy of a file or a directory tree at low cost.
- **Record append** allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individual client’s append.

Architecture

GFS cluster

- Single master
- multiple chunkserver accessed by multiple clients.

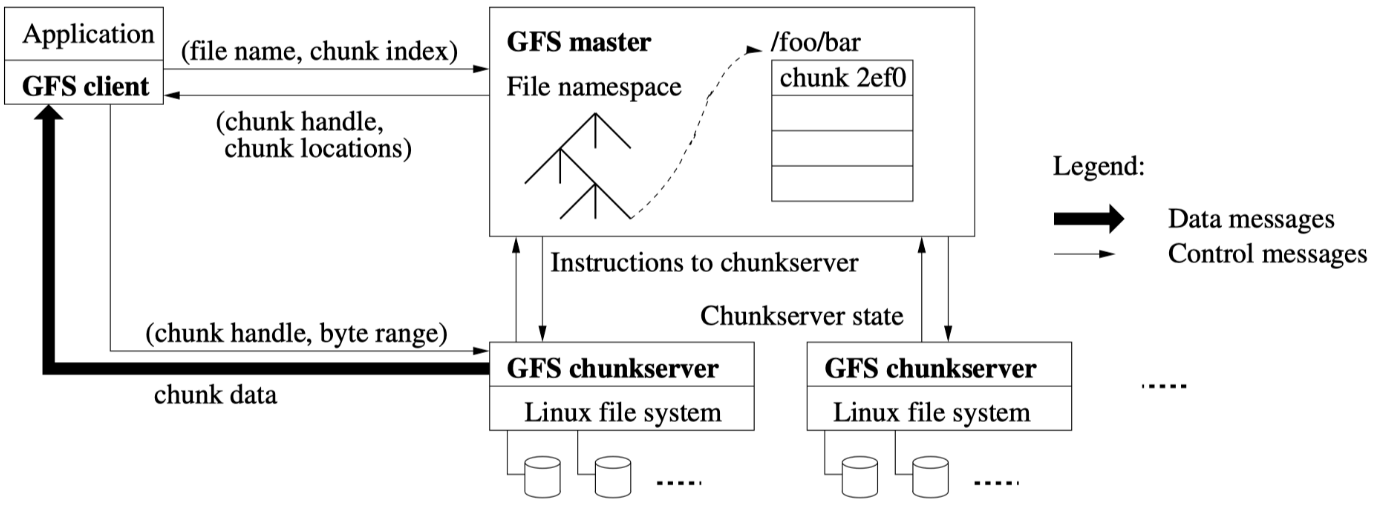

Figure 1: GFS Architecture

Files are **divided into fixed-size chunks**. Each chunk is identified by an **immutable and globally unique 64 bit chunk handle** assigned by the **master** at the time of **chunk creation**.

Chunkservers store chunks on local disks as Linux files and read or write chunk data specified by a **chunk handle and byte range**.

each chunk is replicated on multiple chunkservers(default is 3)

Master maintains all file system metadata.

- namespace,
- access control information,
- the mapping from files to chunks,
- current locations of chunks.

also controls **system-wide activities** such as

- chunk lease management
- garbage collection of orphaned chunks,
- chunk migration between chunkservers

master periodically communicates with each chunkserver in **HeartBeat messages** to give it instructions and collect its state.

Single Master

must minimize its involvement in reads and writes so that it does not become a bottleneck.

interactions for a simple read

1. client translates the file name and **byte offset** specified by the application into a **chunk index** within the file.
2. it sends the master a request containing the file name and chunk index.
3. master replies with the **corresponding chunk handle** and locations of the replicas.
4. The client caches this information using the file name and chunk index as the key.
5. The client then sends a request(chunk handle and a byte range) to one of the replicas, most likely the closest one.
6. Further reads of the same chunk require no more client-master interaction until the cached information expires or the file is reopened

Chunk Size

64MB, extended only as needed(Lazy).

Pros:

- reduces clients' need to interact with the master
- a client is more likely to perform many operations on a given chunk
    - reduce network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time.
    - reduces the size of the metadata stored on the master, thus can keep the metadata in memory.

Cons:

- A small file consists of a small number of chunks, perhaps just one. The chunkservers storing those chunks may become hot spots if many clients are accessing the same file.
- storing such file with a higher replication factor.

Metadata

1. file and chunk namespace
2. mapping from files to chunks,
3. locations of each chunk's replicas. master **asks** each chunkserver about its chunks at master startup and whenever a chunkserver joins the cluster, don't keep a persistent record.

All in memory.

1&2 are also kept by `log` operation stored on the local disk and remote machine.

In-memory Data Structure

Efficient for the master to periodically scan through its entire state, used to implement chunk gc, re-replication for the chunkserver fail-user , chunk migration to balance load and disk space usage.

Chunk Locations

The master does not keep a persistent record of which chunkservers have a replica of a given chunk. It simply polls chunkservers for that information at startup.

update with sending HeartBeat msg.

KEEP IT SIMPLE STUPID.

Operation Log

- Contains a historical record of critical metadata changes.